2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD) | 979-8-3503-2225-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAD57390.2023.10323837

LIM-GEN: A Data-guided Framework for Automated Generation of Heterogeneous Logic-in-Memory Architecture

Libo Shen[1,2],Boyu Long[1,2],Rui Liu[1,3],Xiaoyu Zhang[1,2],Yinhe Han[1],Xiaoming Chen[1,*]
[1]Institute of Computing Technology, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences
[3]Xiangtan University
[*]Corresponding author: chenxiaoming@ict.ac.cn

***Abstract*—Memristor-based logic-in-memory (LIM) is an emerging technology that enables logic operations within memory, making it a promising solution for data-intensive applications. LIM architectures have different types according to where computations are executed, with each type being suitable for specific design objectives and application domains. However, mapping applications to a single LIM mode restricts the full utilization of different LIM modes. In this paper, we propose LIM-GEN, a data-guided framework for automated generation of heterogeneous LIM architectures. To take advantages of different LIM modes, three LIM modes are combined and used as building blocks to create heterogeneous architectures. Given the data-centric nature and large design space, there is an urgent need of developing new EDA tools for synthesizing such LIM architectures. LIM-GEN includes an automatic hardware synthesis flow, which takes behavior-level descriptions as input to generate application-specific architectures and dataflows. During synthesis, data distribution, task allocation and crossbar mapping are optimized through a design space exploration process. We evaluate LIM-GEN in several data-intensive applications and compare the generated heterogeneous architectures with synthesized architectures with a single LIM mode. The experimental results demonstrate significant improvements in latency, area and power consumption, brought by the heterogeneous architectures generated by LIM-GEN.**

***Index Terms*—Logic-in-memory, design automation, memristor, heterogeneous architecture**

I. Introduction

Traditional von Neumann computer architectures require frequent data transfers with memory during computation, which leads to significant performance degradation when dealing with data-intensive applications. This is the so-called *memory wall* bottleneck. Benefitting from the electronic characteristics of non-volatile devices, like memristors, the compute-in-memory (CIM) technology has emerged as a new computing paradigm, which eliminates the need of transferring data from/to the processing units. CIM is considered to be a promising solution to the memory wall challenge.

This work was supported by National Key R&D Program of China (No. 2018YFA0701500), by National Natural Science Foundation of China (Nos. 62122076, 61834006, and 62025404), by Key Research Program of Frontier Sciences, CAS (No. ZDBS-LY-JSC012), by Strategic Priority Research Program of CAS (No. XDB44000000), and by Innovative Project of Institute of Computing Technology, CAS (No. E261040).

Depending on the application scenario and the method of operation, memristor-based CIM architectures can be divided into two main categories. One type is processing-in-memory (PIM) for neural networks, such as ISAAC [1], PRIME [2]. They use memristor crossbars to accelerate analog matrix-vector multiplications. The other type is logic-in-memory (LIM) for general applications which involve lots of logic operations. Different from PIM, the various LIM architectures differ greatly in the way they perform operations. Ref. [3] summarizes and classifies LIM implementations according to the data statefulness, proximity of computation and flexibility.

From the computing mode point of view, LIM can be generally classified into three categories: compute-in-crossbar (CC) [4]–[6], push-to-periphery (PP) [7]–[9] and search-after-storage (SS) [10]–[12]. Each LIM mode has its own applicable scenarios and drawbacks.

CC uses the electronic characteristics of memristors to perform logic operations within crossbars. Both input and output are represented by resistance of memristors, and the result is written directly into an output memristor. All operations are performed within crossbars. This LIM mode significantly reduces data movement. It does not require complex peripheral circuits, making it ideal for fine-grained bit operations with minimized area overhead. However, the large number of memristor write operations results in lower performance and higher energy consumption.

PP depends on the peripheral circuits of memory arrays for logic operations. This mode usually activates multiple memory rows and gets the logic result of activated rows in the peripheral circuits, which provides extremely high parallelism. The currents flowing through the activated input memristors are transferred to the peripheral circuits and the result of some bit-wise logic operation is obtained through modified sense amplifiers (SAs). As the input and output are represented by resistance and voltage, respectively, the output needs to be written back to the crossbar first for future operations. In addition, data in the memory needs to be fully aligned for massively parallel bit-wise logic operations.

SS is a kind of memristor-based FPGAs, where memristors are used to store the truth tables of lookup tables (LUTs), replacing the conventional volatile SRAMs. Both the input and output are represented by voltage. Benefiting from LUTs, this LIM mode helps to implement complex logic operations (e.g., cascaded logics). It needs input and output data to be stored in other locations rather than in the LUTs, which results in data transfers. In addition, the area of peripheral circuits is not small.

Existing LIM architectures only use a single logic mode, thus suffering from the disadvantages of the adopted logic mode, as mentioned above. For example, MAJ [6] and IMPLY [5] belong to the CC category, and PINATUBO [7] is an implementation of PP, while ME [13] belongs to the SS type. In practical applications, different computation tasks have different characteristics and a single logic mode may not fit well. To fully exploit the advantages of the LIM technology, different LIM modes should be combined, and computation tasks should be elaborately mapped to the most suitable logic modes, taking into account the characteristics of both the computation tasks and the logic modes.

In this paper, we propose LIM-GEN, a data-guided framework designed for automated generation of heterogeneous LIM architectures. We utilize software/hardware co-design within this framework to generate power, performance and area (PPA)-optimized LIM architectures. On the hardware side, we combine the three LIM modes to create a heterogeneous LIM architecture, which is used as a template (backbone) for synthesis and mapping. On the software side, we develop a synthesis and mapping flow that automatically converts VHDL behavior descriptions to application-specific LIM architectures. During the synthesis and mapping flow, PPA is optimized through a design space exploration (DSE) process, under given constraints. Different from the conventional logic synthesis and high-level synthesis flows, where data is scheduled according to the locations of computing components, in LIM-GEN, we decide the locations of computation tasks according to data locations to minimize data transfers, so it is a *data-guided* flow.

Our contributions in this paper are listed as follows.

- We propose combing different LIM modes to create heterogeneous LIM architectures, to fully exploit their advantages, enabling different computation tasks to be optimally mapped to the most suitable LIM modes.
- We optimize the existing LIM modes and modularize them to build a rich design space, to support the exploration of the optimal heterogeneous LIM architectures.
- To generate heterogeneous LIM architectures from application behavior descriptions, a synthesis flow is proposed. The synthesis takes VHDL as input and performs a series of optimizations on data distribution, task allocation and crossbar mapping, to generate PPA-optimized LIM architectures.

II. Logic-in-Memory Background

A. LIM Basics

LIM is a technology that uses the electronic characteristics of special memory devices to perform in-situ logic operations within memory. Memristor is an emerging nonvolatile device that has the potential for LIM [14]. The resistance of a memristor can be switched between high resistance state (HRS) and low resistance state (LRS) by applying corresponding voltages to its terminals. Memristors can be used to perform LIM logic operations, with HRS and LRS representing logic 0 and 1, respectively. As a passive device, directly connecting multiple memristors to form crossbar arrays will cause sneak paths [15]. One-transistor-one-memristor (1T1R) is a basic unit commonly used to form an array, as shown in Fig. 1(a).

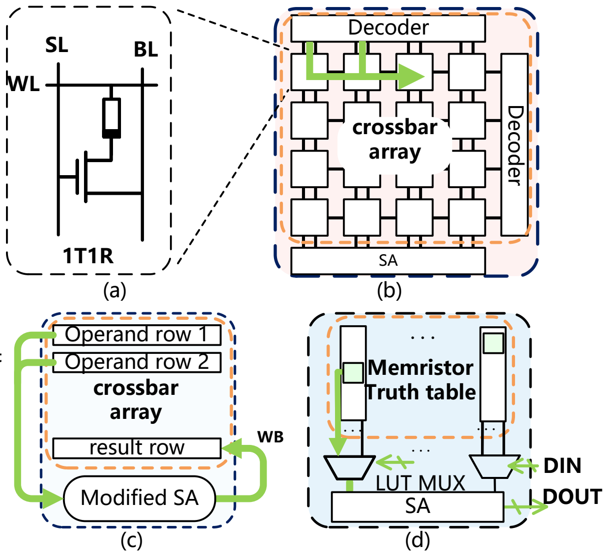


Fig. 1. Overview of LIM. (a) 1T1R unit. SL: select line. WL: word line. BL: bit line. (b) Compute-in-crossbar (c) Push-to-periphery. Act: activation. WB: write back. (d) Search-after-storage. DIN: input. DOUT: output.

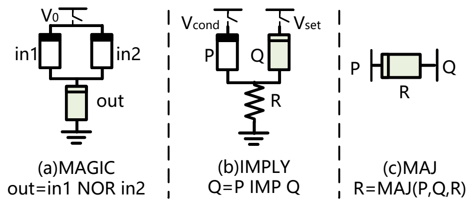


Fig. 2. Three different CC implementations and their basic operations. (a) MAGIC [16]. (b) IMPLY [5]. (c) MAJ [6].

B. Compute-in-Crossbar

This kind of LIM performs "computing while writing". MAGIC [16], MAJ [6] and IMPLY [5] are three typical CC implementations, as shown in Fig. 2. Both input and output are expressed as memristor resistance values. MAGIC [16] does not modify the inputs, which is very suitable for use in LIM, as shown in Fig. 1(b) and Fig. 2(a). Based on MAGIC, FELIX [17] expands its logical operations, enabling NOR, NOT, NAND, MIN, and OR in one step.

C. Push-to-Periphery

The endurance and switching latency of memristors limit the use of CC. To alleviate the problems, LIM using peripheral circuits for computing is proposed, which we call

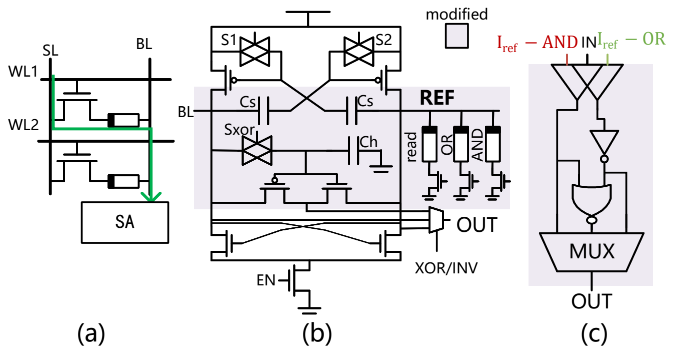


Fig. 3. Modified-SA for PP LIM. (a) 1T1R in crossbar. (b) C-SA [7]. (c) D-SA [9].

PP. The connection between SA and the array is shown in the Fig. 3(a). The sense amplifiers (SAs) in the peripheral circuits are modified to support highly-parallel bitwise logic operations. There are two typical implementations, capacitor SA (C-SA) [7] and double-threshold SA (D-SA) [9], as shown in Fig. 3. They use reference resistors or reference currents for logic operations. Typically two or more memory rows are activated simultaneously for bitwise logic operations, as Fig. 1(c) shows, but they do not deal with operations between bit-vectors in the same row. This makes them inflexible. In addition, inputs of PP are expressed as memristors' resistance, while outputs are expressed as SAs' output voltages. When outputs are re-involved in further computations, they need to be written back to the crossbar array first.

D. Search-after-Storage

Both CC and PP support only a limited number of logic operations. Complex logic must be unrolled, resulting in a large number of intermediate results. CC and PP require frequent writing to memristors and are not suitable for handling complex logic operations. Several works [10], [11], [13] have designed memristor-based FPGAs that can solve these problems well. As Fig. 1(d) shows, memristors store the truth tables of LUTs. The outputs (DOUT) are fed back to the input selection MUXes of the LUTs, so that an $N \times N$ crossbar array can support at most N levels of cascaded logic. In our design, SS is usually used as an acceleration unit for complex logic, with both inputs and outputs represented by voltage.

E. LIM Design Automation

In traditional high-level synthesis, a limited number of arithmetic devices are allocated during the scheduling process, and placement and routing are performed based on the scheduling result [18]. It not applicable for LIM because the routing result is required to schedule communication [19]. In LIM architectures, data are stored and operated in the same area, and storage units play the role of arithmetic devices. The scheduling process becomes "putting data in right places". It means that LIM design automation needs to be guided by data.

The design space for efficient data allocation and leveraging different LIM units is vast, necessitating the use of automated methods to aid in the design process. Once scheduling is completed, the data's specific locations are determined, which involves the technology mapping process of LIM [20], often referred to as crossbar mapping in [21]. An automated generation framework for heterogeneous LIM architectures requires a combination of synthesis and crossbar mapping to achieve optimal PPA results.

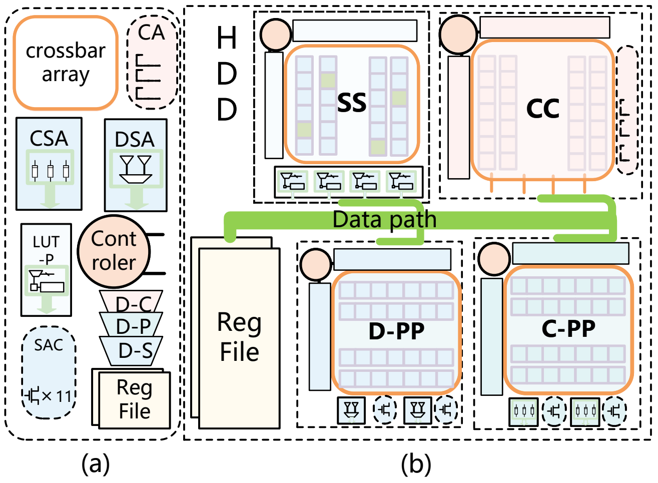


Fig. 4. Overview of heterogeneous architecture template. (a) Hardware library. (b) Heterogeneous architecture. D-C/D-S/D-P: decoders of CC, SS and PP architectures. CA: carry calculation and alignment unit. SAC: sum and carry unit. HDD: hardware description diagram.

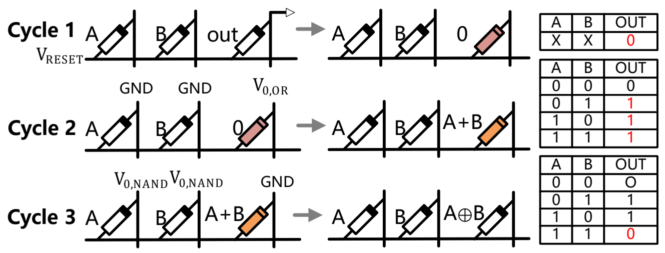
III. Hardware Architecture

This section introduces our proposed heterogeneous LIM architecture template which will be used for automated architecture generation. We split the LIM components into modules and use a graph-based description for the design space. The generality of an architecture can be defined as a combination of flexibility, scalability and compatibility. As mentioned earlier, we have selected three LIM modes. In order to design heterogeneous architectures flexibly and prepare for the design automation process, we optimize the generality of each LIM mode and build a rich design space.

A. Architecture Template Overview

The overview of the heterogeneous LIM architecture template is shown in Fig. 4. Green arrows represent the data path. We store data in crossbar arrays and choose modular functional components to perform required computations. This constitutes a rich design space. Components need to be generic and modular in order to meet various design needs. We will optimize the three LIM modes for generality below.

With the 1T1R crossbar array as the basic building block of crossbar arrays, the peripheral circuits around them can have different components, as Fig. 4 shows. Components marked by the same color belong to the same LIM mode, and different LIM modes can be arbitrarily combined to form different architecture designs. The register file stores immediate values or intermediate results. The three LIM modes need different decoders: in PP decoders need to activate multiple rows for bitwise logic operations, while in CC complex decoders are

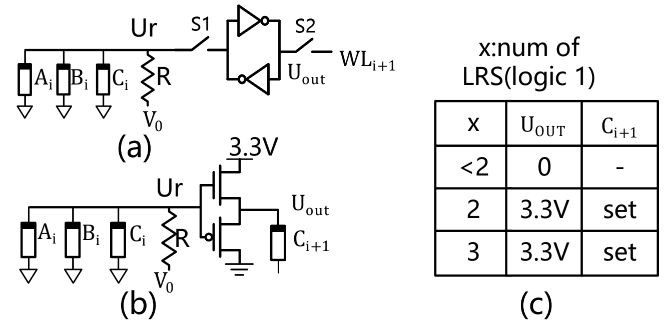


A	B	OUT
X	X	0

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

A	B	OUT
0	0	O
0	1	1
1	0	1
1	1	0

Fig. 5. Steps for implementing 2-bit XOR using FELIX [17].



x	U_{OUT}	C_{i+1}
<2	0	-
2	3.3V	set
3	3.3V	set

Fig. 6. Carry calculation and alignment unit. (a) Our CA. (b) CA without switch [22]. (c) State transition table.

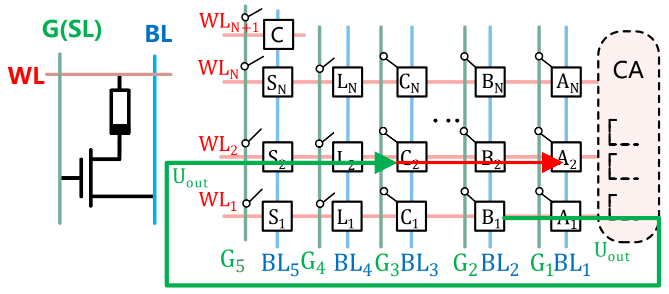


Fig. 7. Our designed CC carry calculation and alignment circuit. WL: word line. BL: bit line. G(SL): select line serves as a gate.

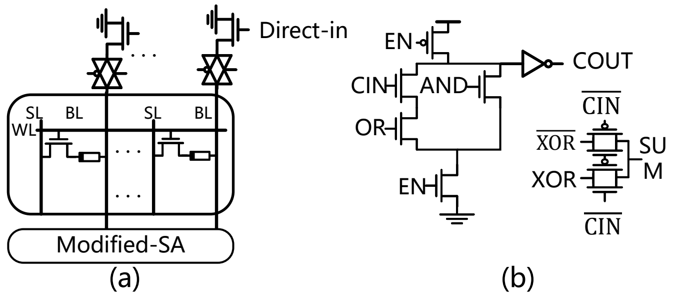


Fig. 8. PP optimization. (a) Direct-input unit of PP. (b) Sum and carry unit [8].

required to provide additional voltage levels to support in-situ logic operations and write back. When an array needs to perform ADD operations, carry- and sum-related circuits can be added to it. In PP, depending on the SA design for logic operations (D-SA or C-SA, see Fig. 3), we have two different implementations of PP: C-PP and D-PP. SA can also add functional components as needed, which we will describe in detail below.

B. CC Carry Calculation and Alignment

CC performs fine-grained logic operations within a crossbar array. It is cumbersome to implement carry alignment when performing multi-bit ADD. We need peripheral circuitry to help with carry alignment. We choose FELIX [17] as the basic method for CC logic operations. When performing an ADD operation, FELIX is based on

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_i, \\ C_{i+1} &= A_iB_i + B_iC_i + A_iC_i = MAJ\{A_i, B_i, C_i\}. \end{aligned} \tag{1}$$

The process of an XOR operation of FELIX is shown in Fig. 5. In order to calculate and align the carry bits, we design a carry calculation alignment unit (CA), as shown in Fig. 6. CA consists of two cross-coupled inverters, two switches (S1 and S2), and a resistor whose resistance R is equal to LRS. S2 in WL_i connects to WL_{i+1}. Fig. 6(b) and Fig. 6(c) show the principle. When the number of LRS among A_i, B_i and C_i is greater than or equal to 2, the output voltage of the inverter will be pull up to 3.3V, and C_{i+1} will be set. Based on this principle, our designed carry calculation and alignment circuit is shown in Fig. 7. The green line represents the calculation process. To avoid changing A_{i+1} and B_{i+1}, as shown by the red line in the figure. We use the CA's two inverters and switches to temporarily store the result, and then continue the transmission after closing G_1 and G_2.

C. PP Write-back Reduction

PP performs bitwise logic operations by modified SAs for multiple activated memory rows. Unlike the "Computing while writing" of CC, storage and execution of PP must be performed serially because of the signal conversion between voltage and resistance. As mentioned above, PP requires the inputs stored in the crossbar array. When there are a lot of intermediate results in the calculation, frequent write back operations seriously affect the performance.

To reduce write back operations, we add a direct-input row in each crossbar array. As Fig. 8(a) shows, a direct-input unit consists of a transmission gate and a transistor. By applying data to the direct-input terminals, it acts as the same function of an activated memory row. The intermediate results in continuous operations or data from other arrays can continue to participate in subsequent operations through the direct-input row without writing back. There is a trade-off between writing data back and reading data to the direct-input units. Latches are needed to store data for the direct-input units.

Similar to CC, PP also has problems with carry calculation and transmission when performing ADD. Ref. [8] proposes a sum and carry unit integrated in each SA, consisting of 11 MOSFETs. As Fig. 8(b) shows, carry signals can be transferred directly between SAs without going through the crossbar array. This unit can also be used as an additional component of C-SA and D-SA to implement fast ADD operations.

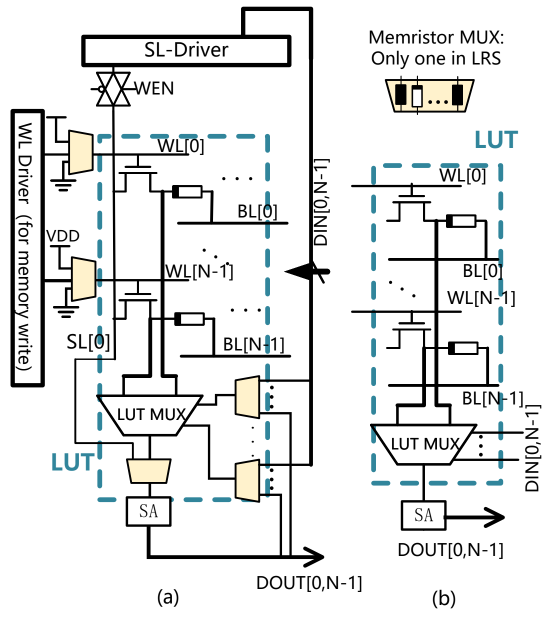


Fig. 9. Memristor-based LUT. (a) ME [13]. (b) LUT in LIM-GEN. Blue box: LUT. DIN: input data. RADDR: row address. CADDR: column address. WEN: write enable.

D. SS Area Optimization

SS uses memristor-based LUTs for logic operations. We modify ME [13] in LIM-GEN. Its circuit diagram is shown in Fig. 9. There are many memristor-based MUXes for implementing reconfigurable logic. In this work, after logic synthesis and crossbar mapping, only one memristor in each MUX will be set to LRS. In other words, the hardware architecture is fixed for the given application, which means that we do not need the reconfigurability feature. We remove the memristor-base MUXes in ME to save area and power overhead. All yellow MUXes in the figure will be changed to wires. For a 64×64 array, the overhead of 25664 memristors can be saved.

Additionally, complex logic may require multiple levels of cascading. In order to modularize the operation, we can use synthesis tools to obtain the number of cascades and the number of LUTs required, and use them to determine the latency and size of the array, which further reduces the area and power.

E. Hardware Library

The hardware library consists of components and operation parameters. From the above component optimizations, we have built various optimized components for different LIM modes. By pre-synthesizing the basic operations of various components, the latency, energy consumption and devices required for the operations can be measured. The hardware library will be used in the LIM-GEN framework to help evaluate the generated architecture.

IV. Design Automation Framework

In this section, we introduce the LIM-GEN framework in detail. The framework automatically converts VHDL behavioral descriptions to heterogeneous LIM architectures based on the architecture template. The automatic generation process includes compilation, synthesis, and crossbar mapping, as well as an integrated design space exploration flow.

A. Overview of LIM-GEN

The overview of LIM-GEN is shown in Fig. 10. The inputs to the framework include a VHDL behavior description, design objectives, and constraints. We first compile the VHDL netlist into the form of a control data flow graph (CDFG) by using the hdlConvertor tool [23], and generate initial control steps for the operations in the CDFG. The design automation process will start from the initial control steps. We use the simulated-annealing algorithm for design space exploration, through synthesis, crossbar mapping, and simulation in the iterations, to obtain an optimized heterogeneous LIM architecture. The overall generation process is an iterative DSE loop. In each DSE iteration, the data locations, the control steps of the operations, and the LIM type of each operation are changed according to the simulated-annealing algorithm. Generally, the design automation has the following two steps.

Synthesis and crossbar mapping: Schedule the operations in the control steps, determine where the data is stored, and select the appropriate hardware for the operations.

Simulation, evaluation and iteration: Use the integrated simulator to obtain performance parameters, and evaluate performance and explore the design space until results converge.

Different from prior LIM logic synthesis works [6], [24] that dismantling applications to fit the designed architectures, in LIM-GEN, we generate application-specific architectures to fit the applications to achieve higher optimization. The locations where the operations are performed are all guided by the data location. In LIM-GEN, users can input area and power constraints to constrain the automatic generation process.

B. Synthesis and Crossbar mapping

In LIM-GEN, synthesis and crossbar mapping are coupled in a single step. The synthesis process is guided by the data and maps data to crossbars. This step generates a hardware architecture based on the CDFG with operations' control steps and determines the location of the data. In this process, data allocation and task allocation will be determined. We first randomly select the storage locations for the data, and generate an initial hardware description diagram (HDD). HDD describes the hardware architecture using the functional components. The LIM architecture can be disassembled into different functional components. We select components from the hardware library according to the design requirements to form a heterogeneous architecture.

We select the execution locations of the operations according to the guidance of the data locations, and add components and create new crossbar arrays during this process.

1) Dependencies: Executing operations in parallel according to their control steps needs to handle dependencies. Dependencies can be classified into three categories: data dependency, control dependency, and structural dependency.

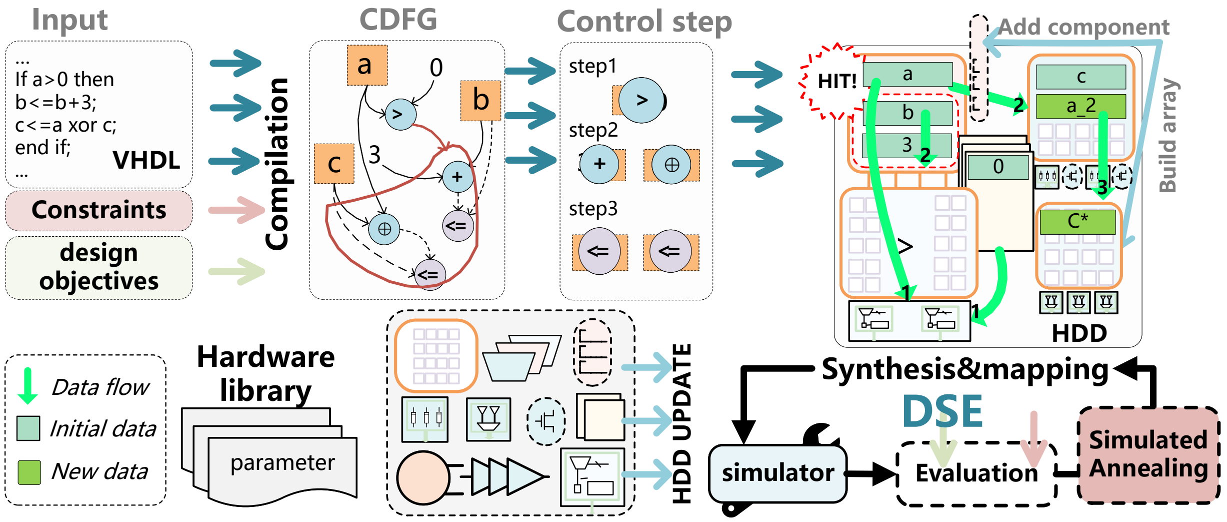


Fig. 10. Overview of LIM-GEN. HDD: hardware description diagram.

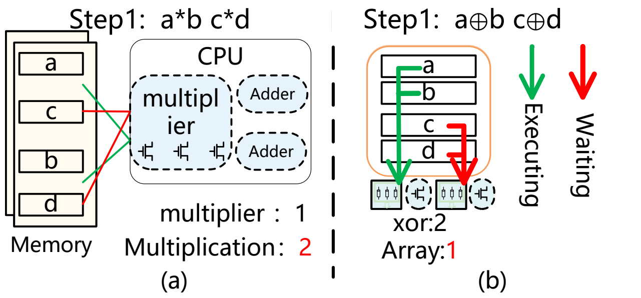


Fig. 11. Structural conflict. (a) Traditional architecture. (b) LIM architecture.

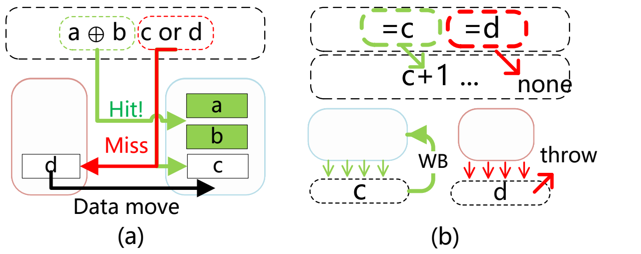


Fig. 12. Select execution array. (a) Hit strategy for calculation nodes. (b) Out-degree strategy for assignment nodes. WB: write back.

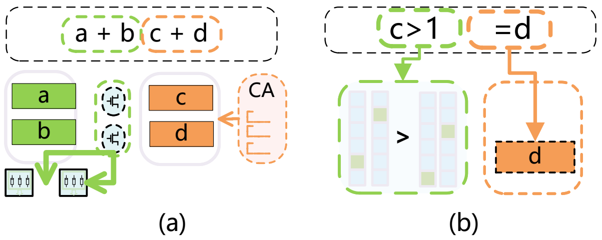


Fig. 13. (a) Add components. CA: carry calculation and alignment unit. (b) Create new array.

Data dependency and control dependency are implied in the CDFG while structural dependency cannot be represented by the CDFG because it is related to hardware. In traditional von Neumann architectures, structural dependency usually represents dependency on computing devices, as shown in Fig. 11(a). In LIM architectures, there is no dedicated computing device. Instead, the structural dependency between operations are determined by the array itself. Consequently, different operations that involve data stored in the same array cannot be executed in parallel, even if they are in the same control step, as shown in Fig. 11(b).

2) Select Execution Array: Nodes in the CDFG can be divided into **calculation** and **assignment** nodes. Calculation nodes can only be run on crossbar arrays, while assignment nodes can also be run by other components such as registers and latches. We design different processing strategies for the two kinds of nodes, as shown in Fig. 12.

Hit strategy for calculation nodes: Before deciding the type of crossbar array to execute, we find the operand. The location at which an operation is executed is influenced by both the locations of the operands and the design objectives. There is a "hit relationship" between the execution array and the operands. As shown in the Fig. 12(a), when two operands are present in the same array, we get a "hit", so the current operation will be decided to execute on this array; otherwise, we get a miss. In the case of a miss, it becomes necessary to retrieve the data from one place to another, causing data movement between arrays. We choose to execute the operation at one of their locations according to the considerations such as waiting time, design target requirements, array type and operation matching. Enhancing the hit rate can significantly boost the overall performance of the architecture.

Out-degree strategy for assignment nodes: In LIM-GEN, all operation results are assigned to variables for future use. As a result, assignment nodes typically have multiple out-degrees. However, when the out-degree of an assignment node is zero, there is no need to store the value in the storage unit, as shown in Fig. 12(b).

3) Add Components and Create New Arrays: The hardware generation process can be described by HDD. During the synthesis of the operations, we add components to the original HDD or create a new array in it. We devise separate strategies for adding components and creating new arrays, as shown in Fig. 13. The component here commonly refers to CA and SAC, which is not included in the initial HDD for area saving. When the current array needs to execute ADD operations, we add it to the array and update the HDD.

In the face of two situations, it is necessary to create a new array. One is when the current array cannot perform the functions, such as performing complex logic. Complex logic means the operations that cannot be supported by CC or PP (e.g., multiplication, comparison or division). The second is that the storage space of the current array has been used up. We decide a priori to use SS for complex logic. For highly-parallel bit-wise simple logic operations, we choose PP for the newly created crossbar, while for other logic operations, we choose CC. The crossbar types will be randomly changed during DSE to find the best configurations. It is worth noting that determination process of the array type requires some randomness, because the impact of the execution of an operation on the overall performance is unknown and hard to evaluate. They will be explored during DSE. After creating a new crossbar array, we update the HDD accordingly.

C. Simulation, Evaluation and Iteration

LIM-GEN integrates a behavioral simulator. Based on the hardware library built in Section III, we can get the current execution time, energy consumption and hardware information after each step of operation is executed.

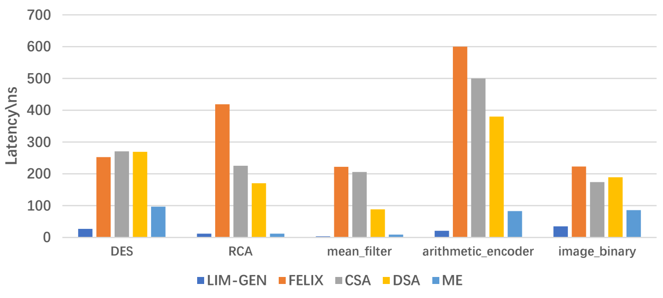
After the above steps, we can get the power consumption and latency according to the simulator. The design information of the current architecture can be obtained according to the HDD. We use the simulated-annealing algorithm for DSE, in which the current architecture is randomly perturbed by randomly changing the data locations, array type and control steps of operations. The objective function is the user given design objective. The constraint can be total power or area consumption. Under the constraint, we can explore the optimal heterogeneous architecture design for optimized latency, power consumption or area.

V. Evaluation

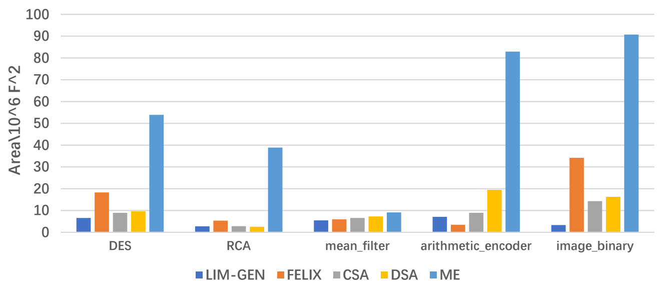
In this section, we evaluate the performance of the generated LIM architectures by LIM-GEN.

TABLE I
Memristor parameters [25].

Parameter	Value	Parameter	Value
HRS	300kΩ	$Area_m$	$4F^2$
LRS	1kΩ	$Area_{1T1R}$	$6F^2$
V_{set}	-1.5V	T_{switch}	1ns
V_{reset}	0.3V	E_{switch}	0.13pJ



(a) Latency



(b) Area

Fig. 14. Results and comparisons under power constraint.

A. Experiment Setup

Before simulation, we build a library of hardware parameters and pre-synthesize the basic operations of the three LIM modes. We use HSPICE with VTEAM [26] and PTM 45nm [27] to simulate the parameters of CC and the 1T1R unit, and Design Compiler to obtain the parameters of digital components. The basic parameters of memristor are based on Ref. [25], as shown in Table I.

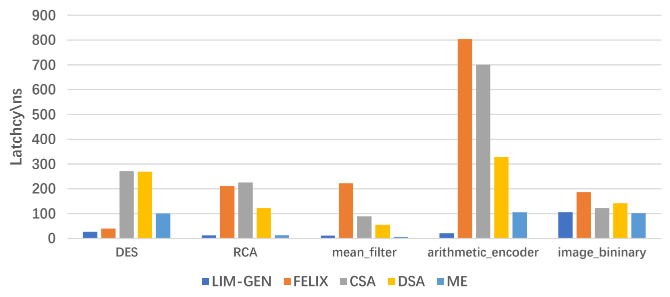
To validate the performance of the heterogeneous architectures generated by LIM-GEN, we select several common general logic applications, including DES encryption, RCA encryption, mean filtering, image binarization processing, and arithmetic encoding algorithm. These applications involve a significant number of Boolean logic operations, making them ideal candidates for acceleration using LIM architectures.

The performance parameters of the heterogeneous architectures are simulated by the integrated simulator of LIM-GEN. To demonstrate the advantages of combining different LIM modes, we also use LIM-GEN to create several single-mode baselines. That is to say, we restrict the synthesis and mapping process to use only a single LIM mode, i.e., FELIX (CC logic) [17], C-SA [7], D-SA [9] (PP logic) or ME (SS logic) [13], to obtain single-mode baselines, and compare them separately with the heterogeneous architectures generated by LIM-GEN.

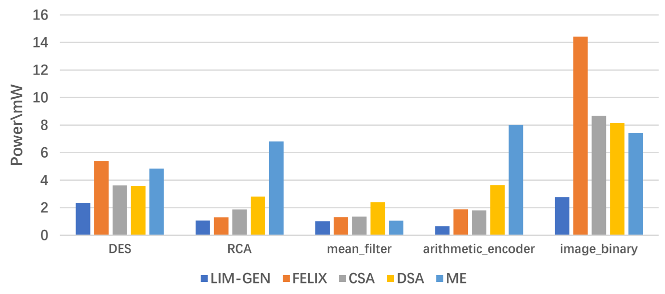
Runtime: LIM-GEN is implemented in C++ and takes approximately an hour to synthesize a LIM architecture.

B. Performance Evaluation

LIM-GEN is a powerful tool that can optimize various design objectives while adhering to specific design constraints.



(a) Latency



(b) Power

Fig. 15. Results and comparisons under area constraint.

In evaluation, each design is exclusively optimized for only one specific metric, while the other metrics are required to remain within reasonable ranges. In our cases, we perform design optimization considering two crucial constraints: area and power consumption.

Under the constraint of the total power consumption, the optimization results of LIM-GEN for latency and area are depicted in Fig. 14(a) and Fig. 14(b), respectively.

Fig. 14(a) demonstrates the optimization of latency using LIM-GEN. It reveals that LIM-GEN achieves speedups of $13.9\times$, $11.1\times$, $8.6\times$, and $2.65\times$ compared with the original implementations of FELIX, C-SA, D-SA, and ME, respectively. The execution speed of FELIX (CC) is relatively slow due to the inclusion of numerous memristor write operations. C-SA and D-SA (PP) exhibit better parallelism, but their performance can also be affected by data write-back operations. ME (SS) does not require data conversion during operations, but the data movements between storage units and LUTs introduce additional latency overhead.

Fig. 14(b) presents the optimization results of LIM-GEN for total area. It demonstrates substantial area optimizations achieved by LIM-GEN, with improvements of $85.7\times$, $3.1\times$, $4.5\times$, and $20.4\times$ compared with FELIX, C-SA, D-SA, and ME, respectively. FELIX utilizes a relatively small area, but when processing complex logic, a large number of intermediate results contribute to significant area overhead. ME, on the other hand, has a larger area footprint due to the utilization of a large number of memristor-based MUXes.

Under the constraint of the total area consumption, the optimization results of LIM-GEN for latency and power consumption are presented in Fig. 15(a) and Fig. 15(b), respectively.

Fig. 15(a) illustrates the optimization of latency using LIM-GEN. It demonstrates speedups of $6.5\times$, $3.9\times$, $1.9\times$, and $1.1\times$ achieved by LIM-GEN compared with FELIX, C-SA, D-SA, and ME, respectively.

On the other hand, Fig. 15(b) shows the optimization results of LIM-GEN for power consumption. LIM-GEN achieves power savings of $6.1\times$, $3.8\times$, $4.1\times$, and $4.4\times$ compared with FELIX, C-SA, D-SA, and ME, respectively. Notably, FELIX exhibits higher power consumption due to the elevated write overhead of memristors when numerous write operations are involved.

The inclusion of CA units in CC, SAC units and direct-input units in PP, and the removal of the memristor-based MUXes in SS contribute to significant optimizations in both latency and area.

In the case of CC, the addition of the CA units accelerates the calculation of complex logic, reducing the overhead associated with memristor write operations when computing intermediate results. This enhancement improves the overall execution speed of CC and helps to minimize latency.

Similarly, for PP, the inclusion of the direct-input units and the SAC units enables operations to be performed with reduced write-back of intermediate results. This improvement enhances the parallelism and efficiency of PP, resulting in increased speed and reduced power consumption.

In the case of SS, the removal of the memristor-based MUXes leads to significant area optimization. The reduction in area utilization enhances the overall efficiency of the design.

By incorporating these modifications, LIM-GEN effectively optimizes latency, area and power consumption. The improvements in the hardware components, such as the CA units, direct-input units, and removal of the memristor-based MUXes, contribute to enhanced performance, reduced power consumption, and efficient resource utilization in the design process.

In addition to hardware optimization, LIM-GEN combines the advantages of the three LIM modes, puts data in the best locations through design space exploration, and uses the appropriate LIM modes to perform operations, achieving significant PPA improvements.

VI. Conclusion

In this paper, we propose a design automation framework capable of designing heterogeneous LIM architectures. The framework combines synthesis and mapping techniques with data-guided hardware generation. We evaluate the performance of LIM-GEN using a variety of general logic applications, such as encryption, image processing, and encoding. Comparing heterogeneous LIM architectures designed by LIM-GEN with four individual LIM architectures, we observe significant improvements in latency, area, and power consumption. This demonstrates the conclusion that a single LIM mode may not be the best, and instead, different LIM modes should be combined for different logic operations.

References

[1] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 14–26.

[2] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 27–39.

[3] J. Reuben, R. Ben-Hur, N. Wald, N. Talati, A. H. Ali, P.-E. Gaillardon, and S. Kvatinsky, "Memristive logic: A framework for evaluation and comparison," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.

[4] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[5] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "'memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, Apr 2010. [Online]. Available: https://doi.org/10.1038/nature08940

[6] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for rram-based in-memory computing using majority-inverter graphs," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 948–953.

[7] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[8] D. Reis, M. Niemier, and X. S. Hu, "Computing in memory with fefets," in *Proceedings of the International Symposium on Low Power Electronics and Design*, ser. ISLPED '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3218603.3218640

[9] M.-F. Chang, S.-J. Shen, C.-C. Liu, C.-W. Wu, Y.-F. Lin, Y.-C. King, C.-J. Lin, H.-J. Liao, Y.-D. Chih, and H. Yamauchi, "An offset-tolerant fast-random-read current-sampling-based sense amplifier for small-cell-current nonvolatile memory," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 3, pp. 864–877, 2013.

[10] Y. Y. Liauw, Z. Zhang, W. Kim, A. E. Gamal, and S. S. Wong, "Non-volatile 3d-fpga with monolithically stacked rram-based configuration memory," in *2012 IEEE International Solid-State Circuits Conference*, 2012, pp. 406–408.

[11] S. Tanachutiwat, M. Liu, and W. Wang, "Fpga based on integration of cmos and rram," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 11, pp. 2023–2032, 2011.

[12] J. Cong and B. Xiao, "Fpga-rpi: A novel fpga architecture with rram-based programmable interconnects," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 4, pp. 864–877, 2014.

[13] X. Chen, L. Yin, B. Liu, and Y. Han, "Merging everything (me): A unified fpga architecture based on logic-in-memory techniques," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–2.

[14] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, 1971.

[15] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 476–488.

[16] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[17] S. Gupta, M. Imani, and T. Rosing, "Felix: Fast and energy-efficient logic in memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–7.

[18] S. Gerez, *Algorithms for VLSI Design Automation*. Wiley, 1998.

[19] J. Yu, R. Nane, I. Ashraf, M. Taouil, S. Hamdioui, H. Corporaal, and K. Bertels, "Skeleton-based synthesis flow for computation-in-memory architectures," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 2, pp. 545–558, 2020.

[20] D. Bhattacharjee, Y. Tavva, A. Easwaran, and A. Chattopadhyay, "Crossbar-constrained technology mapping for reram based in-memory computing," *IEEE Transactions on Computers*, vol. 69, no. 5, pp. 734–748, 2020.

[21] V. Tenace, R. G. Rizzo, D. Bhattacharjee, A. Chattopadhyay, and A. Calimera, "Said: A supergate-aided logic synthesis flow for memristive crossbars," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 372–377.

[22] X. Fu, Q. Li, W. Wang, H. Xu, Y. Wang, W. Wang, H. Yu, and Z. Li, "High-speed memristor-based ripple carry adders in 1t1r array structure," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 9, pp. 3889–3893, 2022.

[23] M. Orsak, "hdlconvertor," https://pypi.org/project/hdlConvertor/, 2021.

[24] S. Shirinzadeh and R. Drechsler, "Logic synthesis for in-memory computing using resistive memories," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018, pp. 375–380.

[25] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature Nanotechnology*, vol. 8, no. 1, pp. 13–24, Jan 2013. [Online]. Available: https://doi.org/10.1038/nnano.2012.240

[26] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, 2015.

[27] ASU, "Predictive technology model," http://ptm.asu.edu/, 2011.